Computational Ex1

Noam and Maor

2024-07-04

Computation ex1

Question 1

Part 1

First, the **floating point system** is represented by the following formula:

$$Fl(x) = \left\{ \pm \frac{m}{2^t} \cdot 2^e | \quad 2^{t-1} \leq m < 2^t, e_{min} \leq e \leq e_{max} \right\} \cup \{0\}$$

Consequently, the **maximum** positive number we can get in our floating point system is:

$$\frac{(2^t - 1)2^{e_{max}}}{2^t} = 2^{e_{max}} \cdot (1 - 2^{-t})$$

Part 2

The **smallest** positive number we can get in our floating point system is:

$$\frac{(2^{t-1})}{2^t} 2^{e_{min}} = 2^{e_{min-1}}$$

Part 3

When talking about **Normalized** and **Denormalized** system, one might want to represent them first:
Normalized:

$$F'(x) = \{ \pm 0.d_1 ..., d_n \cdot 2^e | d_1 = 1, d_2, ...d_n \in \{0, 1\}, e_{min} \leq e \leq e_{max} \}$$

Denormalized:

$$F''(x) = \{ \pm 0.d_1 ..., d_n \cdot 2^e | d_1, d_2, ...d_n \in \{0, 1\}, e_{min} \leq e \leq e_{max} \}$$

Thus, the **smallest** positive Denormalized number we can get:

$$\frac{1}{2^t} 2^{e_{min}} = 2^{e_{min} - t}$$

Part 4

The **smallest** positive integer that **can't** be represented in our floating point system is:

$$N = \lfloor N_{max} \rfloor + 1 = \lfloor 2^{e_{max}} \cdot (1 - 2^{-t}) \rfloor + 1$$

Indeed, We can represent all positive integers in the range of $[fl_{min}(x), fl_{max}(x)]$.

Part 5

Example 1:

We're going to fix t = 3, $e_{max} = 100, e_{min} = -100$, and we'll choose:

$e_1 = 3, e_2 = 4, m_1 = 4, m_2 = 4$

Our chosen e and both m satisfy:

$$Fl(x) = \left\{ \pm \frac{m}{2^t} \cdot 2^e | \quad 2^2 \leq m < 2^3, e_{min} \leq e \leq e_{max} \right\}$$

Thus, the number we're going to represent is:

$$Fl_1(x) = Fl_2(x) = \pm \frac{4}{2^3} \cdot 2^3 = \pm 4$$

We're going to sum them up:

$$Fl_1(x) + Fl_2(x) = 4 + 4 = 8 = m_1 + m_2 = \frac{4}{2^3} \cdot 2^4$$

Now, in our floating point system the sum is exact:

$$\frac{4}{2^3} \cdot 2^3 + \frac{4}{2^3} \cdot 2^3 = \frac{4}{2^3} \cdot 2^4 = 4 + 4 = 8$$

Overall, we represented 2 numbers in our floating point system and the floating sum is exact.

Example 2:

We're going to fix t = 3, $e_{max} = 100, e_{min} = -100$, and we'll choose: $e_0 = 1, m_1 = 7, m_2 = 4$

Our chosen e and both m satisfy: 2

$$Fl(x) = \left\{ \pm \frac{m}{2^t} \cdot 2^e | \quad 2^2 \leq m \leq 2^3, e_{min} \leq e \leq e_{max} \right\}$$

So we get:

$$Fl_1(x) = \pm \frac{7}{2^3} \cdot 2^1 = \pm \frac{7}{4} Fl_2(x) = \pm \frac{4}{2^3} \cdot 2^1 = \pm 1$$

Finally when multiplying them we get $\frac{7}{4} \cdot 1 = \frac{7}{4}$

In our floating point system the multiplication is exact with m = 7 which is in the range of $2^2 \leq m \leq 2^3$.:

$$\frac{7}{2^3} \cdot 2^1 \cdot \frac{4}{2^3} \cdot 2^1 = \frac{7}{2^3} \cdot 2^1$$

Question 2

Part 1

The **smallest** positive integer we **can't** represent:

In **IEEE single** floating point system is:

$$N_{single} = \lfloor fl_{Smax} \rfloor + 1 = \lfloor (2^{t_{single}} - 1) \cdot 2^{e_{smax}-t} \rfloor + 1 = \lfloor (2^{24} - 1) \cdot 2^{128-24} \rfloor + 1 = \lfloor (2^{24} - 1) \cdot 2^{104} \rfloor + 1$$

In **IEEE double** floating point system is:

$$N_{double} = \lfloor fl_{Dmax} \rfloor + 1 = \lfloor (2^{t_{double}} - 1) \cdot 2^{e_{dmax}-t} \rfloor + 1 = \lfloor (2^{53} - 1) \cdot 2^{1024-53} \rfloor + 1 = \lfloor (2^{53} - 1) \cdot 2^{971} \rfloor + 1 = (2^{53} - 1) \cdot 2^{971} + 1$$

Part 2

Adding 1.0 will not change the value. When adding 1.0 to N_{double}, We add a small number with 2^{-971} in the mantissa. Thus, due to precision limitation, because the relative error:

$$\frac{|N_{double} + 1 - N_{double}|}{N_{double}} = |\frac{(2^{53} - 1) \cdot 2^{971} + 1 + 1 - (2^{53} - 1) \cdot 2^{971} - 1}{(2^{53} - 1) \cdot 2^{971} + 1}| < \quad 2^{-971} \quad < \epsilon_{mach}$$

Therefore, the value will remain the same.

We added a test in the **end** of the pdf file to show that the value remains the same.

Question 3

Part 1

As we can see, when performing dot product we multiply each element in the vector by the corresponding element in the other vector and sum them up.

This results in *p multiplications **and** p-1 additions**:

$$< u, v >= \sum_{i=1}^{p} u_i \cdot v_i$$

When analyzing the numerical error we get for i = 1,2...p:

$$fl(\sum_{i=1}^{p} u_i v_i) = (\cdot(\cdot(\cdot(u_1 v_1)(1 + \delta_1^*) + (u_2 v_2)(1 + \delta_2^*))(1 + \delta_1^+) + ... + (u_p v_p)(1 + \delta_p^*)) \cdot (1 + \delta_+^{p-1})$$

To sum it all up we get:

$$fl(u_1 v_1 + u_2 v_2 + ... + u_n v_n) = \tilde{S} = u_1 v_1 (1 + \delta_1^*) \prod_{i=1}^{p-1} (1 + \delta_i^+) + \sum_{i=2}^{p} u_i v_i (1 + \delta_i^*) \prod_{j=i-1}^{p-1} (1 + \delta_j^+)$$

Following this, the relative error is as follows:

$$\frac{|\tilde{S}-S|}{|S|} = \frac{|u_1v_1(1+\delta_1^*)\prod_{i=1}^{p-1}(1+\delta_i^+) + \sum_{i=2}^{p} u_iv_i(1+\delta_i^*)\prod_{j=i-1}^{p-1}(1+\delta_j^+) - \sum_{i=1}^{p} u_iv_i|}{|\sum_{i=1}^{p} u_iv_i|} =$$

$$= \frac{|u_1v_1\cdot(p\cdot\epsilon_{mach}+O(\epsilon_{mach}^2)) + \sum_{i=2}^{p} u_iv_i\cdot(p\cdot\epsilon_{mach}+O(\epsilon_{mach}^2))|}{|\sum_{i=1}^{p} u_iv_i|} = \frac{|\sum_{i=1}^{p} u_iv_i|(p\cdot\epsilon_{mach}+O(\epsilon_{mach}^2))|}{|\sum_{i=1}^{p} u_iv_i|} \approx p\cdot\epsilon_{mach}$$

$$\leq |\sum_{i=1}^{p} u_iv_i|(\max_{i\in[1,n]}\sum_{i=1}^{p-1}\delta_i^* + \delta_j^+ + O(\epsilon^2)) \approx (p-1)\epsilon_{mach} + \epsilon_{mach} + O(\epsilon^2) = p\cdot\epsilon_{mach}$$

Part 2

The upper bound for the relative error using the **FMA** operation is $(p-1)\cdot\epsilon_{mach}$, because if FMA we have only one rounding error in the end of the calculation:

$$fl(x\pm y) = (x\pm y)(1+\delta_{\pm})$$

We can see that the numerical error is slightly smaller than the error we got in the previous part but it's negligible for large p.

Part 3

One example is **binary tree** summation that reduces the number of operations to $O(\log n)$, by tree depth, instead of n. Summing this way instead of summing consequently all elements, decreases the number of times we multiply by $1+\delta^+$ and thus reduces the error.

Question 4

Part 1

We have here 2 ways to compute the variance:

1. The **first** way is to calculate the sample variance by the formula:

$$SE^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2$$

2. The **second** way is to calculate the sample variance by the formula:

$$SE^2 = \frac{1}{n}(\sum_{i=1}^{n} x_i^2) - \bar{x}^2$$

Looking at the first formula, we can see that we have n multiplications and n-1 additions, which results in a total of 2n-1 operations.

Conversely, the second formula has n multiplications and n additions, which results in a total of 2n operations.

Moreover, when looking at formula (2) we can see that we perform the following steps:

1. Calculate the sum of the squares of the elements:

This results in squaring and summing big numbers when dealing with [$10^{9,10}$9+1,10^9+2] and thus we might lose information:

$$\sum_{i=1}^{n} x_i^2 = (10^9)^2 + (10^9+1)^2 + (10^9+2)^2$$

Due to precision limitation, we lose information when summing big numbers.

2. Calculate the mean of the elements and then squaring:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{10^9 + 10^9 + 1 + 10^9 + 2}{3}$$

$$\bar{x}^2 = \left(\frac{10^9 + 10^9 + 1 + 10^9 + 2}{3}\right)^2$$

Again, we square and sum big numbers.

3. Subtract the mean from the sum of the squares

Now, when we perform these steps, we create a new error that we don't have with the first formula:

Let's look at the relative error for IEEE single:

$$\frac{|(10^9+1)^2 - 10^{18}|}{|(10^9+1)^2|} = \frac{10^{18} + 2\cdot 10^9 + 1 - 10^{18}}{(10^9+1)^2} = \frac{2\cdot 10^9 + 1}{(10^9+1)^2} \leq 10^{-18} \leq \epsilon_{mach_{single}}$$

Moreover, using it in IEEE Double we get big error as we've seen in our python code, which is implemented with IEEE Double.

Part 2

In our example the first formula is much more accurate than the second one, because we subtract each i observation from the mean and then square and sum it all up.

Consequently, we deal with smaller numbers in between steps and thus we lose less information because of precision limitations.

Question 5

Part 1

A function is Lipschitz continuous with constant L, when for all x,y in the domain of the function we have:

$$|f(x) - f(y)| \leq L \cdot |x - y|$$

When a function is Lipschitz continuous, it means that the function is bounded by a linear function with a slope of L. This means that the function is not too steep and doesn't change too much in a small interval.

Let f be a function with bounded derivative in the interval (a,b), then f is Lipschitz continuous with constant L. Let's look at constant L: if f is differentiable in the interval (a,b), then the derivative of f is bounded in the interval (a,b) by M, then f is Lipschitz continuous with constant L = M. We know from a theorom in the second lecture that $\hat{k} = ||J_f(x)||$, So the Lipschitz constant is related to the absolute condition number.

On the other hand, if a function is Lipschitz continuous with constant L, and we take a differentiable function then the derivative is bounded by L according to lagrange theorem. Thus, the absolute condition number is bounded by L for almost every x in the domain of the function, because the absolute condition number is the norm of the Jacobian matrix.

Part 2

We added a python file to the **end** of the pdf file.

Part 3

Using repeated multipication to compute $p_2(x)$ can indeed reduce the numerical accuracy, especially when dealing with large numbers and high degree polynomials. The reason for this is that when we use the pow function or similar ones in different languages (like R) we use a well defined and executed way to operate the power function. This way is more accurate than repeated multiplication, because it uses a more sophisticated algorithm to compute the power of a number. Furthermore, we suggest that with implicit multiplication we add numerous rounding errors at each multiplication, which can lead to a significant loss of accuracy, as opposed to using the pow function.

Question 6

Part 1

Let's look at the relative condition number of a tan(x):

tan(x) is differentiable in the interval $(-\frac{\pi}{2} + \pi n, \frac{\pi}{2} + \pi(n+1))$

We know that:

$$k(x) = \frac{||J_{tan}(x)||}{\frac{||\tan(x)||}{||x||}}$$

Because we're in 1 dimension, the Jacobian matrix of tan(x) is:

$$||J_{tan}(x)|| = \frac{1}{cos^2(x)} = 1 + tan^2(x)$$

$$||f(x)|| = |tan(x)|$$

$$||x|| = |x|$$

Thus, we'll compute the relative condition number:

$$k(x) = \frac{1 + tan^2(x)}{|\frac{tan(x)}{x}|} = \frac{|x|(1 + tan^2(x))}{|tan(x)|} = \frac{|x|}{|tan(x)|} + |x||tan(x)|$$

The function tan is a periodic function with period pi, meaning that $\tan(x) = tan(x + n\pi)$ for any integer n.

As x becomes very large the calculation of tan(x) will result in higher odds for numerical errors, because of precision limitations.

When looking at the cyclic feature of tan(x) this means that even a tiny error in x can lead to significant change in the value of tan(x).

Thus, computing tan(x) for a very large number can lead to a numerical error.

Moreover, the relative condition number of tan(x) in Big numbers can be unstable, because a small change in large x can lead to a significant change in the value of tan(x), and to a large k(x)

Part 2

Thus, we suggest solving tan for $x = 10^{100} \mod \pi$.

Because of the cyclic nature of the tan function, we can solve this problem by computing tan for a much smaller number, which is the residual of $10^{100} \mod \pi$.

This way we can avoid the complications of computing tan(x) for a large number, which increases the odds for numerical errors because of precision limitations of the computer.

In order to perform these high number mods we need a library that can handle such large numbers, a library with high precision such as mpmath in python.

Question 7

Part 1

In this question we're using a random perturbation to find an empirical estimate of the condition number.

This is being done with a straightforward simulation:

1. Drawing random $\delta_x \sim N(0, \sigma^2 I)$ with standard error as small as we want.
2. Calculating the condition number with respect to the perturbation: $k = \frac{||f(x+\delta_x)-f(x)||}{||\delta_x||}$
3. Iterating the process with a large number of times and calculating the maximum condition number.
4. Comparing the empirical condition number with the true condition number.

We would expect to get a condition number close to the true condition number, because the perturbation is close to 0 and the condition number is defined as:

$$k = \lim_{\delta \to 0} \sup_{\triangle x : ||\triangle x|| < \delta} \frac{||f(x + \triangle x) - f(x)||}{||\triangle x||}$$

We can see that in our simulation we try to get delta as close to 0 as possible, and thus we should get a condition number close to the true condition number.

For example, let $f(x) = x^2$, then the true condition number is 2x, as we've seen in lecture 2. We'll take x = 1:

$$\hat{k}(x) = lim_{\delta \to 0} \sup_{\triangle x : ||\triangle x|| < \delta} \frac{||f(x + \triangle x) - f(x)||}{||\triangle x||} = lim_{\delta \to 0} \sup_{\triangle x : ||\triangle x|| < \delta} \frac{|(1 + \delta_x)^2 - 1|}{|\delta_x|} = 2 + \delta_x = 2$$

Furthermore, we can compute the empirical condition number:

$$\hat{k} = \max_{\delta_x \sim N(0,\sigma^2 I)} \frac{||f(x+\delta_x) - f(x)||}{||\delta_x||} = \max_{\delta_x \sim N(0,\sigma^2 I)} \frac{|(1+\delta_x)^2 - 1|}{|\delta_x|} = \max_{\delta_x \sim N(0,\sigma^2 I)} \frac{2\delta_x + \delta_x^2}{|\delta_x|} = \max_{\delta_x \sim N(0,\sigma^2 I)} 2 + \delta_x$$

Thus, we can get a lower or an upper bound for the condition number, depends on the sampled random variable.

Part 2

In an **added** python file at the **end** of the pdf file.

Question 8

Part 1

Let f(x) = 2x, We'll check if the function is backward stable:

$$\tilde{f}(x) = 2x(1+\delta_x)(1+\delta_x^*) = 2\tilde{x} = 2x(1+\delta_x)(1+\delta_x^*) = f(\tilde{x})$$

And we get:

$$\tilde{x} = x(1+\delta_x)(1+\delta_x^*)$$

Let's compute the backward relative error:

$$\frac{|\tilde{x}-x|}{|x|} = \frac{|x(1+\delta_x)(1+\delta_x^*)-x|}{|x|} = \frac{|x|}{|x|} \cdot |(1+\delta_x)(1+\delta_x^*)-1| = |\delta_x + \delta_x^* + O(\epsilon^2)| \leq 2\epsilon_{mach} + O(\epsilon^2) = O(\epsilon_{mach})$$

Consequently, the function is backward stable.

Part 2

Let f(x) = x^2, We'll check if the function is backward stable:

$$\tilde{f}(x) = (x(1+\delta_x)x(1+\delta_x))(1+\delta_x^*) = x^2(1+\delta_x)^2(1+\delta_x^*) = f(\tilde{x}) = (\tilde{x})^2 \Rightarrow$$

And we get:

$$\tilde{x} = x(1+\delta_x)\sqrt{1+\delta_x^*}$$

Let's compute the backward relative error:

$$\frac{|\tilde{x}-x|}{|x|} = \frac{|x(1+\delta_x)\sqrt{(1+\delta_x^*)}-x|}{|x|} = \frac{|x|}{|x|} \cdot |(1+\delta_x)\sqrt{(1+\delta_x^*)}-1| = |\sqrt{(1+\delta_x^*)} + \sqrt{(1+\delta_x^*)} \cdot \delta_x - 1| \leq$$

After multiplying by the conjugate we get:

$$|\frac{(\sqrt{(1+\delta_x^*)}-1)\cdot(\sqrt{(1+\delta_x^*)}+1)}{(\sqrt{(1+\delta_x^*)}+1)}| = |\frac{\delta_x^*}{\sqrt{(1+\delta_x^*)}+1}| \leq \frac{|\delta_x^*|}{2} \leq \frac{\epsilon_{mach}}{2}$$

$$\leq 2\epsilon_{mach} + 0.5\epsilon_{mach} + O(\epsilon^2) = O(\epsilon_{mach})$$

So we got that this function is backward stable.

Part 3

Let $f(x) = \frac{x}{x} = 1$, We'll check if the function is forward stable:

$$\tilde{f}(x) = \frac{x(1+\delta_x)}{x(1+\delta_x)}(1+\delta_x^{frac}) = 1+\delta_x^{frac} > 1 = f(\tilde{x}), \quad \text{for} \quad \delta_x^{frac} \neq 0$$

So our function is not backward stable. Let's check if it's stable:

$$\text{We'll take:} \quad \frac{|\tilde{x}-x|}{|x|} = O(\epsilon_{mach}), \quad \frac{|\tilde{f}(x)-f(x)|}{|f(x)|} = \frac{|1+\delta_x^{frac}-1|}{|1|} = |\delta_x^{frac}| = O(\epsilon_{mach})$$

So this function is stable.

Part 4

Let $f(x) = x - x = 0$, We'll check if the function is backward stable:

$$\tilde{f}(x) = (x(1+\delta_x) - x(1+\delta_x))(1+\delta_x^{sub}) = 0 = f(\tilde{x}), \text{ for every } \tilde{x}, x$$

So we can choose:

$$\tilde{x} \in (\quad x(1-\delta_x^{sub}), (x(1+\delta_x^{sub}) \quad)$$

and we'll get:

$$\frac{|\tilde{x}-x|}{|x|} = \frac{|x(1-\delta_x^{sub})-x|}{|x|} = \frac{|x|}{|x|}\cdot|(1-\delta_x^{sub})-1| = |\delta_x^{sub}| = O(\epsilon_{mach})$$

So this function is backward stable.

Part 5

Let $e = \sum_{k=1}^{\inf} \frac{1}{k!}$, We'll check if the sum is backward stable:

First of all, we don't have an input for this algorithm so:

$$f(\tilde{x}) = e \neq \tilde{f}(x)$$

So we don't have backward stability.

We'll check if it's stable:

The algorithm is stable if for every x we have:

$$\frac{|\tilde{x}-x|}{|x|} = O(\epsilon_{mach}), \frac{|\tilde{f}(x)-f(x)|}{|f(x)|} = O(\epsilon_{mach})$$

But, for each $k \in N$, When computing the relative error we get:

$$\frac{|\tilde{f}(x)-f(x)|}{|f(x)|} = \frac{|e(1+\delta_k^+)^{k-1}(1+\delta_k^{div})^{k-1}(1+\delta_k^*)^k - e|}{|e|} = \frac{|e|}{|e|}|(1+\delta_k^+)^{k-1}(1+\delta_k^{div})^{k-1}(1+\delta_k^*)^k - 1| =$$

$$= |1-1+(k-1)\delta_k^* + (k-1)\delta_k^+ + k\cdot\delta_k^{div} + O(\epsilon_{mach}^2)| \neq O(\epsilon_{mach})$$

So the algorithm is not stable

Part 6

Like before, The algorithm is not stable, even though when summing from right to left a lot of the accumulated errors are canceled out, because the accumulated sum is much smaller in the first steps as opposed to summing from left to right. Therefore, summing right to left, we get less elements that are smaller than the epsilon machine and thus the error is smaller.

Conversely, the error is still not bounded by the epsilon machine.

Q2

Part 2

In [242…
```
import numpy as np

# Define the floating point constants
N_single = (2**24 - 1) * 2**128 / 2**24
N_single_test = N_single + 1.0

N_double = (2**53 - 1) * 2**1024 / 2**53
N_double_test = N_double + 1.0

# Print the results in a formatted manner
print("Floating Point Representation Analysis")
print("--------------------------------------")
print(f"N_single: {N_single}")
print(f"N_single_test N_single + 1.0: {N_single_test}")
print()
print(f"N_double: {N_double}")
print(f"N_double_test: N_double + 1.0: {N_double_test}")
print()
print("Observations:")
print("1. The value of N_single_test (N_single + 1.0) is the same as N_single, we expected otherwise.")
print("2. The value of N_double_test (N_double + 1.0) is the same as N_double, due to precision limitations"
```

```
Floating Point Representation Analysis
--------------------------------------
N_single: 3.4028234663852886e+38
N_single_test N_single + 1.0: 3.4028234663852886e+38

N_double: 1.7976931348623157e+308
N_double_test: N_double + 1.0: 1.7976931348623157e+308

Observations:
1. The value of N_single_test (N_single + 1.0) is the same as N_single, we expected otherwise.
2. The value of N_double_test (N_double + 1.0) is the same as N_double, due to precision limitations
```

In [47]:
```
N_double-N_single**7 # The same!
```

Out[47]:
```
1.7976931348623157e+308
```

In [220…
```
N_double -N_double/2 # Different!
```

Out[220]:
```
8.988465674311579e+307
```

Q4

Part 1

In [214…
```
def variance_1(sample):
    n = len(sample)
    mu = np.mean(sample)
    return sum((sample-mu)**2)/n
```

In [225…
```
def variance_2(sample):
    n = len(sample)
    mu = np.mean(sample)
    return np.mean(sample**2) - mu**2
```

The python function:

In [ ]:
```
np.var(np.array([10**9,10**9 + 1,10**9 + 2]))
```

In [215…
```
variance_1(np.array([10**9,10**9 + 1,10**9 + 2]))
```

Out[215]:
```
0.6666666666666666
```

In [226…
```
variance_2(np.array([10**9,10**9 + 1,10**9 + 2]))
```

Out[226]:
```
-1.0000000029182744e+18
```

This works for 10^4!

In [235…
```
variance_2(np.array([10**4,10**4+ 1,10**4 + 2]))
```

Out[235]:
```
0.6666666716337204
```

Q5

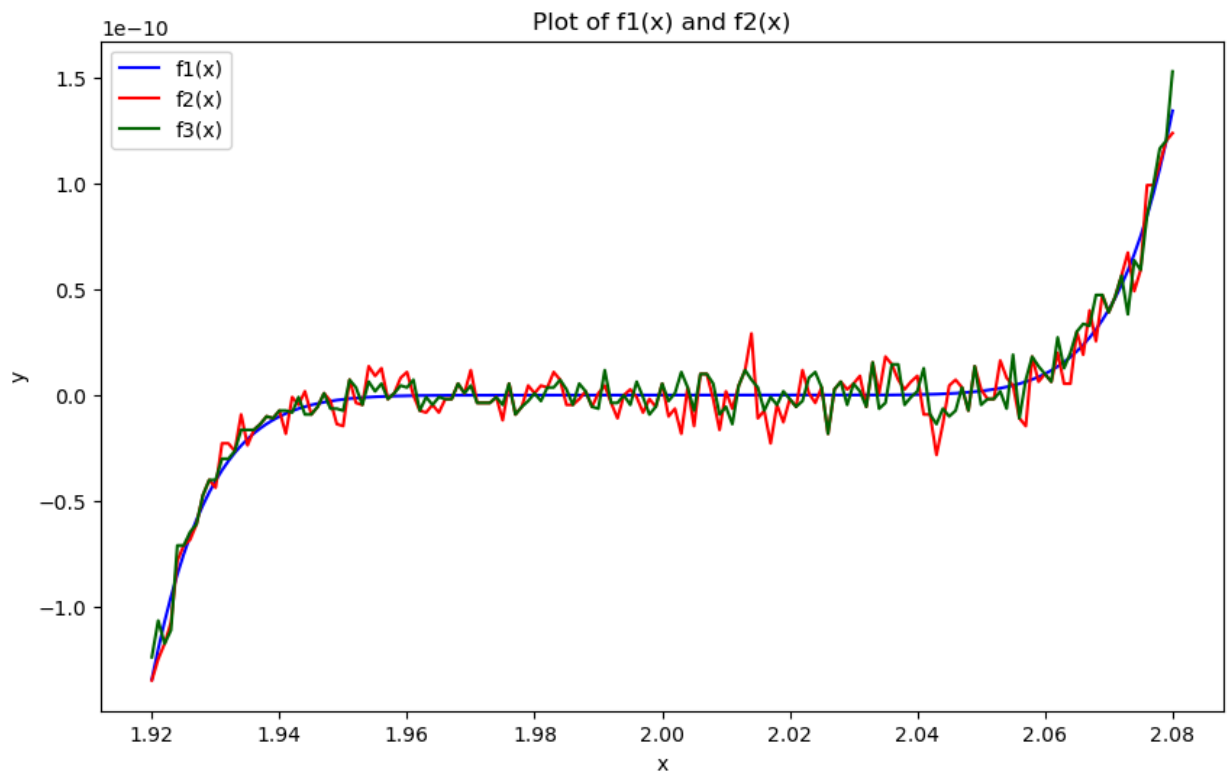
Part 3

In [21]:
```
import numpy as np
import random as random
import math
from scipy.stats import norm
import pandas as pd
import concurrent.futures
```

In [29]:
```
def f1(x):
    return (x - 2)**9

def f2(x):
    return x**9 - 18*x**8 + 144*x**7 - 672*x**6 + 2016*x**5 - 4032*x**4 + 5376*x**3 - 4608*x**2 + 2304*x - 5

def f2_pow(x):

    return (pow(x, 9) - 18 * pow(x, 8) + 144 * pow(x, 7) - 672 * pow(x, 6) +
            2016 * pow(x, 5) - 4032 * pow(x, 4) + 5376 * pow(x, 3) -
            4608 * pow(x, 2) + 2304 * x - 512)

def f2_numpy(x):
    return (np.power(x, 9) - 18 * np.power(x, 8) + 144 * np.power(x, 7) - 672 * np.power(x, 6) +
            2016 * np.power(x, 5) - 4032 * np.power(x, 4) + 5376 * np.power(x, 3) -
            4608 * np.power(x, 2) + 2304 * x - 512)

def f3(x):
    x9 = x * x * x * x * x * x * x * x * x
    x8 = x * x * x * x * x * x * x * x
    x7 = x * x * x * x * x * x * x
    x6 = x * x * x * x * x * x
    x5 = x * x * x * x * x
    x4 = x * x * x * x
    x3 = x * x * x
    x2 = x * x

    return(x9 - 18 * x8 + 144 * x7 - 672 * x6 + 2016 * x5
            - 4032 * x4 +5376 * x3 - 4608 * x2
           + 2304 * x - 512)
```

In [171…
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore", "use_inf_as_na")
# Define the functions


# Generate the vector
vector = np.arange(1.92, 2.08, 0.001,)

# Create the plot
plt.figure(figsize=(10, 6))
sns.lineplot(x=vector, y=f1(vector), label='f1(x)', color='blue')
sns.lineplot(x=vector, y=f2(vector), label='f2(x)', color='red')
sns.lineplot(x=vector, y=f3(vector), label='f3(x)', color='darkgreen')


# Add labels and title
plt.xlabel('x')
plt.ylabel('y')
plt.title('Plot of f1(x) and f2(x)')
```

```
plt.legend()

# Show the plot
plt.show()
```



Q6

Part 2

Python default:

In [236…
```
10**100 % np.pi

np.tan(10**100 % np.pi)
```

Out[236]:
```
0.929036363690337
```

Using the most popular high precision package:

In [179…
```
#Lets compute it accurately:

import mpmath

# Compute the remainder of 10^100 divided by pi
remainder = mpmath.fmod(mpmath.power(10, 100), mpmath.pi)

# Compute the tangent of the remainder
tan_value = mpmath.tan(remainder)

# Print the result
print(tan_value)
```

```
0.929036363690337
```

Using Decimal, we get a different result:

In [187…
```
from decimal import Decimal, getcontext
import math

# Set precision high enough to handle large numbers
getcontext().prec = 110

# Define the large number and pi
```

```
large_number = Decimal(10**100)
pi_value = Decimal(math.pi)

# Compute the remainder of the large number divided by pi
remainder = large_number % pi_value

# Compute the tangent of the remainder using math.tan
tan_value = math.tan(float(remainder))

# Print the result
print(tan_value)
```

```
-0.6026139368359317
```

Using sympy we get this result:

In [188…

```
import sympy as sp

# Define the large number and pi
large_number = 10**100
pi_value = sp.pi

# Compute the remainder of the large number divided by pi
remainder = large_number % pi_value

# Compute the tangent of the remainder
tan_value = sp.tan(remainder)

# Print the result
print(tan_value.evalf())
```

```
0.401231961990814
```

Using gmpy2 we got:

In [198…

```
import gmpy2

# Set precision high enough to handle large numbers
gmpy2.get_context().precision = 113

# Define the large number and pi
large_number = gmpy2.mpz(10**100)
pi_value = gmpy2.const_pi()

# Compute the remainder of the large number divided by pi
remainder = gmpy2.fmod(large_number, pi_value)

# Compute the tangent of the remainder
tan_value = gmpy2.tan(remainder)

# Print the result
print(tan_value)
```

```
1.25628730882732466357736346989382957
```

SO, which one is accurate? generally, mpmath is used way more frequently so we'll go with 0.929036363690337

Q7

Part 2

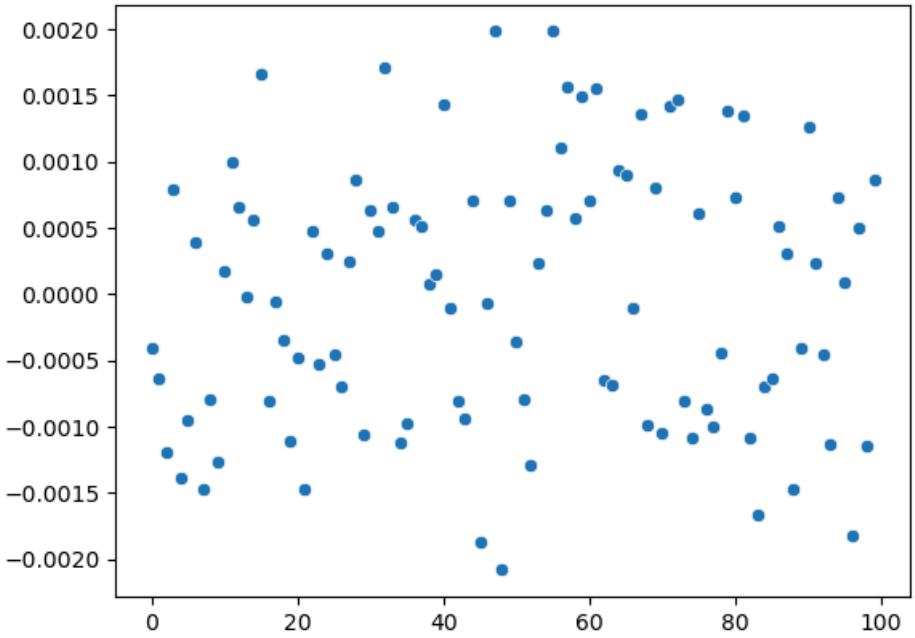
Let's first plot the deltas:

In [237…

```
data = np.random.normal(loc=0, scale=0.001, size=100)  # 100 values with mean 100 and standard deviation 10

# Create an index for the x-axis
index = np.arange(len(data))

# Create a scatter plot
sns.scatterplot(x=index, y=data)
```

Out[237]:

```
<Axes: >
```



In [167…

```
def fun1(x):
    return np.sin(x)

def fun2(x):
    return np.cos(x)

def fun3(x):
    return -x

def fun4(x):
    return 1

def fun5(x):
    return (x-2)**9

def fun6(x):
    return 9 * (x-2)**8

def sim_minus_x(x, n):
    vec = []
    delta = np.random.normal(0, 2**(-23), n)
    for i in range(n):
        est = abs(fun3(x + delta[i]) - fun3(x)) / abs(delta[i])
        vec.append(est)
    return max(vec)

def true_cond_minus_x(x):
    return fun4(x)

def sim_x_power_9(x, n):
    vec = []
    delta = np.random.normal(0, 2**(-23), n)
    for i in range(n):
        est = abs(fun5(x + delta[i]) - fun5(x)) / abs(delta[i])
        vec.append(est)
    return max(vec)

def true_cond_x_power_9(x):
    return fun6(x)
# Example usage
```

In [139…

```
x1 = 0.5
x2 = 0.0085
x3 = 50
n = 10000
```

In [168…

```
# -x:

sim_cond_1 = sim_minus_x(x1, n)
true_condi_1 = true_cond_minus_x(x1)
sim_cond_2 = sim_minus_x(x2, n)
true_condi_2 = true_cond_minus_x(x2)
sim_cond_3 = sim_minus_x(x3, n)
```

```
true_condi_3 = true_cond_minus_x(x3)

#(x-2)^9


sim_cond_x1 = sim_x_power_9(x1, n)
true_cond_x1 = true_cond_x_power_9(x1)
sim_cond_x2 = sim_x_power_9(x2, n)
true_cond_x2 = true_cond_x_power_9(x2)
sim_cond_x3 = sim_x_power_9(x3, n)
true_cond_x3 = true_cond_x_power_9(x3)
```

-x

In [169…
```
{'sim_cond1': sim_cond_1, 'true_condi1':true_condi_1 ,
 'sim_cond_2': sim_cond_2, 'true_condi_2': true_condi_2,
'sim_cond_3': sim_cond_3, 'true_condi_3': true_condi_3 }
```

Out[169]:
```
{'sim_cond1': 1.0000000003096956,
 'true_condi1': 1,
 'sim_cond_2': 1.0000000000259464,
 'true_condi_2': 1,
 'sim_cond_3': 1.0000049767029187,
 'true_condi_3': 1}
```

(X-2)^9

In [170…
```
{'sim_cond1': sim_cond_x1, 'true_condi1':true_cond_x1 ,'sim_cond_2': sim_cond_x2,
 'true_condi_2': true_cond_x2,
'sim_cond_3': sim_cond_x3, 'true_condi_3': true_cond_x3 }
```

Out[170]:
```
{'sim_cond1': 230.6602463857345,
 'true_condi1': 230.66015625,
 'sim_cond_2': 2226.820506558115,
 'true_condi_2': 2226.819395831708,
 'sim_cond_3': 253613532088340.4,
 'true_condi_3': 253613523861504}
```